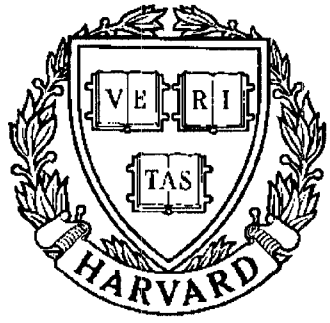
TECHNICAL RESEARCH REPORT

SYSTEMS RESEARCH CENTER



Supported by the National Science Foundation Engineering Research Center Program (NSFD CD 8803012), the University of Maryland, Harvard University, and Industry

A Graphical Simulation Management System

by A. Teolis

TR 91-75

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
1991		**00-00-1991 to 00-00-1991**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
A Graphical Simulation Management System	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **University of Maryland,Systems Research Center,College Park,MD,20742**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**43**	

A Graphical Simulation Management System

A. Teolis *

Systems Research Center
University of Maryland
College Park, MD., 20742

Keywords: 3D Graphics, Dynamical Simulation, Interactive, Object Oriented, Software

Abstract

In this paper we describe the design and implementation of a system for the management of complex interactive graphical dynamical simulations. The main objective of the system is to provide tools which enable a graphical simulation to be built with minimum effort. These tools include routines for building interfaces (panels consisting of buttons, sliders, type in boxes, etc.), arranging the simulation workspace, describing complex 3D objects in terms of predefined (or user defined) primitives, and providing objects with properties used in conjunction with lighting models.

The need for such a management system is supported by the facts that both the development effort and the extent of machine-specific knowledge needed to successfully implement an arbitrary graphical simulation is significant. It is the aim of the Graphical Simulation Management System (GSMS) to reduce these burdens and place the power of 3D simulation within easy reach.

We have presented several examples of dynamical simulations in which the power and diversity of the management system is demonstrated.

*This research was supported in part by the AFSOR University Research Initiative Program under grant AFSOR-90-0105 and by the National Science Foundation's Engineering Research Centers Program: NSFD CDR 8803012.

1 Introduction

Simulation has become a staple of system design. In the case that the system under simulation is amenable to a 3D graphical representation, the ability to graphically represent the result of a complex dynamical system simulation is an appealing prospect. Typically, the implementation of a graphical simulation is a formidable technical endeavor requiring a fairly in depth knowledge of specific machine hardware/software as well as a significant amount of development time. It is the intent of the Graphical Simulation Management System (GSMS) to relax both of these requirements. By providing tools for

(i) building interfaces (panels consisting of buttons, sliders, type in boxes, etc.),

(ii) arranging the simulation workspace,

(iii) describing complex 3D objects in terms of predefined (or user defined) primatives, and

(iv) endowing objects with properties used in conjunction with lighting models,

GSMS does in fact prove to reduce significantly both the development effort and extent of machine specific knowledge required of the simulation builder.

Although the current implementation of the management system is in the first stages of development, the breadth of simulation scenarios which it already supports is significantly vast. Examples are presented in section 4. All examples and the management system itself are written in the C programming language. It is assumed throughout that the reader is familiar with C; an excellent reference is [1].

2 Overview of the System

GSMS exists as a C library archive containing routines which may be called from an application program. It is the function of the application program to describe the 3D object models and to set up the simulation workspace. Exactly how this is accomplished is the major topic of this paper.

Also left to the application is the task of providing GSMS with updates of the dynamical system being simulated. Although GSMS does no internal differential equation solving, it does support interaction between the simulation supplied by the application program and the application user. For example one could set up an application such that its user may easily set or reset certain simulation parameters (via graphical control panels). After setting the parameters, the user may then continue to watch the affect of those changes on the evolution of the system.

Along with providing a general description of GSMS, it is the intent of this paper to explain how a simulation builder would create a GSMS application program.

Several example application programs are detailed in the Appendix. A typical application follows the control flow depicted in Figure 1. There are essentially two processes which are incorporated into the main loop of an application: (i) numerical computation of the current graphical state, and (ii) the display of its graphical representation.

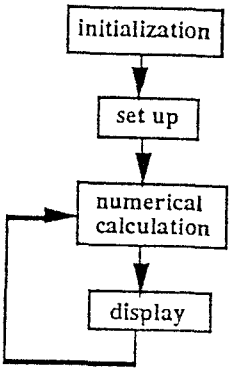


Figure 1: *Typical application control flow*

Routines to perform the first of these tasks, numerical calculation, must be provided by the application. For example, in the case that the system under simulation is the control system

$$\begin{cases} \dot{x} = f(x,u) \\ y = g(x) \end{cases} \quad x \in \mathbb{R}^n, \quad u \in \mathcal{U} \tag{2.1}$$

where u represents the control input, $\mathcal{U}$ the set of admissible controls and $f(\cdot)$ and $g(\cdot)$ describe the system, it is necessary to provide numerical solutions to Equation 2.1.

Depending on its complexity, the required numerical computation may or may not be done at actual execution time. In the case that the computation is extensive a better strategy might be to compute solutions 'off-line' and simply use the graphical simulation system as a display tool. Alternatively one may wish to employ a distributed processing scheme. This sort of implementation is discussed extensively in [2].

3 Anatomy of GSMS

GSMS can be described in terms of several main components:

1. panel manager,
2. display manager,
3. 3D model manager, and

4. lighting, materials/color manager.

Before a detailed account is given we first present a brief overview of each component's function.

It is the panel manager which is responsible for the interactiveness of the application. Through it the application user communicates with the simulation, i.e. input. A panel is simply a window containing a collection of graphical input 'actuators'. An actuator may be a slider controling some simulation parameter or a button which initiates an action when depressed. There are many different types of actuators (see Section 3.1 below).

If the panel manager provides organization for input then certainly the display manager supplies organization for output. Its main responsibility is the management of complex multi-window simulations. The application program specifies precisely the simulation workspace via the display manager.

The 3D model manager is the power plant of GSMS. It is the manager which eases the description and manipulation of complex dynamical 3D objects. Simply stated a complex dynamical object is one which consists of connections of multiple subobjects. Included in the model manager is a list of 3D primatives (spheres, cylinders, etc.) which may serve as the most basic subobject in a complex object description.

An essential component of realism is added via the lighting, materials/color manager. This manager allows the application to easily set up lighting models and endow objects with material properties. Material properties may be associated at the time of object inception via the 3D model manager.

Each manager's function is described in extensive detail below. The reader is refered to the coded examples in the Appendix for the bare technical details.

3.1 Panel Manager

Developed at NASA Ames Research Center [1] the 'Panel Library' [3] is a set of routines for building interfaces. Included in this package are tools for creating panels or windows which consist of various controls such as buttons, sliders, toggles, type-in boxes, etc. Such a package is an invaluable asset for building graphical user interfaces.

Although the 'Panel Library' provides extensive interface capabilities and flexibility, its incorporation into an application can be tedious and time consuming. Hence, the need for a 'panel manager' emerges. With the aid of the panel manager the task of creating interfaces with the application user is greatly simplified.

[1] In collaboration with Sterling Software; the 'Panel Library' is distributed freely.

Below the various building blocks or actuators of an interface are described in detail. Currently, the panel manager only supports a subset of the actuators offered by the 'Panel Library': (i) buttons, (ii) sliders, (iii) type-ins, as well as a new type of actuator not present in the 'Panel Library', (iv) variable (type-ins).

3.1.1 Creating a Panel

new_panel(char *str);

Creation of a panel is invoked by this routine which initializes a panel with the name indicated by the *str* argument. All further calls to other actuator placement routines will place the corresponding actuator in this new panel. Without further calls to other building block routines the panel will remain empty.

In the case that is desirable to have multiple panels, **new_panel()** must be called before each new panel description. For example to create an interface with two panels the structure of the code would look like the skeleton shown in Table 1.

```
new_panel("panel 1"); /* start describing first panel */
                      ⋮
      calls to building routines
                      ⋮
new_panel("panel 2"); /* start describing second panel */
                      ⋮
      calls to building routines
                      ⋮
```

Table 1: Panel creation structure

3.1.2 Actuator Placement

Actuators are arranged in columns automatically by GSMS. The action of GSMS is to stack actuators in columns as they are created. That is the first actuator created in a panel is placed in its lower left hand corner; while all other actuators are placed on top of the previous one until the routine **new_column()** is called.

new_column()

When building a panel, **new_column()** tells GSMS to start placing actuators in a new column. This new column appears to the right of all previous columns.

3.1.3 Buttons

Button actuators perform specified actions when they are selected or 'pressed' with the mouse. There are two types of buttons: (i) toggle buttons and (ii) action buttons. Toggle buttons have associated with them a variable, while action buttons (simply refered to as buttons) have associated with them a procedure or function.

make_toggle_button(char *str, float *ptr)

The *str* argument is a pointer to a string which will appear below the button as a label. When a toggle button is selected its associated variable, **ptr*, is logically toggled. That is if **ptr* is set to logical FALSE (TRUE respectively) depressing its button actuator will toggle its value to logical TRUE (FALSE respectively).

For example, a toggle button might be used to control whether an axis is displayed or not in a three dimensional scene.

make_button(char *str, void (*function))

The *str* argument is a pointer to a string which will appear below the button as a label. When an action button is selected its associated routine, *function()*, is invoked.

For example, an action button may used to initiate a control algorithm.

3.1.4 Sliders

A slider is used to control the value of a variable by associating its numerical value with a position on a line segment. Once end points of the line segment are specified the value of the variable is assigned according to the sliders position with linear interpolation between the end points.

make_slider(char *str, float min, float max, float *var, void (*function))

The *str* argument is a pointer to a string which will appear below the slider as a label. Arguments *min* and *max* provide the range of the slider and *var* is a pointer to the variable to be affected. In many cases it may be desirable to have a procedure or function associated with a slider which is invoked whenever the value of its variable is changed. The argument *function* specifies a pointer to such a procedure. If it is not desired to associate a function to the slider, the predefined special argument **null_function** should be passed.

For example, a slider might be used to control the value of a certain parameter, say θ, of a function, $f_\theta(\cdot)$ that is being plotted. Each time the value of θ is changed it is necessary to redraw the plot. Associating a routine that causes the plot to be redrawn with the slider controlling θ accomplishes this.

3.1.5 Type-ins

In order to facilitate textual input from the application user, type-ins produce a box which when selected allow keyboard input.

make_type_in(char *str, char *c_ptr, void (*function))

The *str* argument is a pointer to a string which will appear below the type-in as a label; while *c_ptr* is a pointer to the character string in which the application user's input is to be stored; and *function* is a pointer to a routine which is invoked when the application user hits the return key.

As an example a type-in box would be useful to specify the name of a file which is to be read.

3.1.6 Variables

Variables are type-in like boxes whose input is expected to be a floating point value. Associated with a variable box is in a floating point variable. Upon selection its value is set to that specified by the application user. Additionally a procedure may be associated with the variable box.

make_variable(char *str, float *ptr, void (*function))

The *str* argument is a pointer to a string which will appear below the variable box as a label; while *ptr* is a pointer to the floating point variable which is affected by the application user's input; and *function* is a pointer to a routine which is invoked when the application user hits the return key.

3.1.7 Reports

Reports provide a way for the application to communicate with its user. When called with an associated message, a window is created and displayed with that message as its content along with a button labeled 'OK'. The action of the 'OK' button is to destroy its parent window.

report_window(char *str)

The **str* argument is a pointer to a string which will appear as the message of the newly created report window.

3.2 Display Manager

For a graphical simulation using multiple windows, keeping track of the data flow to each window can become quite cumbersome without the aid of a high level manager. Consider a typical case where it is desired to construct a graphical simulation which consists of several windows. Say for instance one is interested in building a simulation with (i) a 3D graphical representation of the 'real' system,

(ii) one or more 2D (or perhaps 3D) plot(s) of some pertinent data, (iii) a window in which the user can construct an input function with the mouse interactively, and (iv) various panels constructed using the panel manager of the previous section. Clearly one can see that the situation can become quite tedious without the incorporation of some structure, i.e. a manager. This is the charge of the display manager.

The display manager consists of a list of routines which oversee the creation and display of multiple window applications. Since the objects contained in the windows are of a dynamic nature it is necessary that the application program ask the display manager to update its windows often enough, e.g. once every iteration through the main control loop (Figure 1). The display manager routine which performs this update is called **do_window_stuff()**.

A pictorial representation of this routine's action is displayed in Figure 2. When called **do_window_stuff()** cycles once through every Graphic in the application performing the required action for that Graphic's type. In addition it automatically takes care of updating the panels.

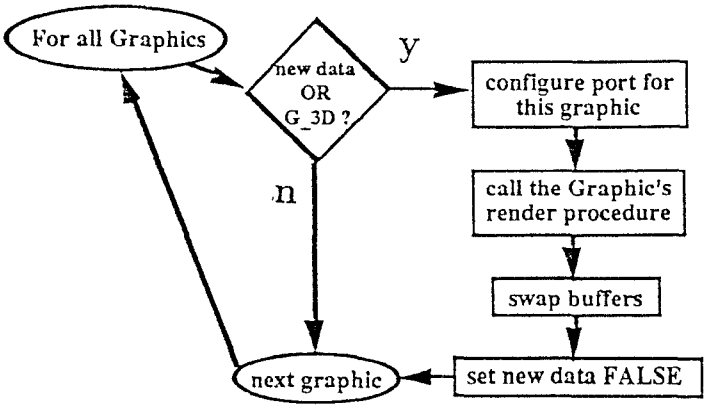


Figure 2: Display manger action (a call to **do_window_stuff()**)

3.2.1 The 'Graphic' Structure

In providing structure to the problem of multiple window display it is necessary to introduce a generalized notion (or more precisely a data structure) capable of representing window types of interest. As a matter of nomenclature this concept of a generalized window is called a *Graphic*. Major elements of the *Graphic* data structure are shown in a partial listing of its declaration given in Table 2.

Each of these fields of the data structure will be explained in the following. Depending on the type of *Graphic* some fields of its data structure will be applicable and some will not.

```
typedef struct graphic {
   int type, color, new_data;
   char title[STRLEN];
         ⋮
   Graph graph;
   Rect *R;
   int interpolation;
         ⋮
   int (*render)();
} Graphic;
```

Table 2: Graphic data structure

title This field of the structure holds the name or title of the window. Each display window's title appears in the upper left hand corner during application execution.

type As has already been discussed, it is desired that the simulation application display several different types of windows. It is the *type* field of the data structure which is assigned to contain window type information. Table 3 lists the basic types of Graphics available. The most basic delineation within Graphic types is that between 2D and 3D. Contents of all 2D Graphic types are assumed to be data plots.

display type	*description*
G_3D	3 dimensional perspective projection
CONSTRUCTOR	application user buildable x-y plots
ARRAY	array versus index plots
ARRAY_vs_ARRAY	(y) array versus (x) array plots
LINK_LIST	(x, y) pairs stored in link-list
INCREMENTAL	update plot only (no erase)

Table 3: Graphic types

new_data, color, and *interpolation* In the case of a G_3D Graphic the new_data, color and interpolation fields are ignored. In all other cases the content of the Graphic is assumed to be a 2D plot. In this case the *new_data* field is a boolean which determines whether the contents of the Graphic have changed. This field is typically set by the application as a cue to the display manager that a certain 2D plot must be redrawn. Interpolation type between adjacent points in the plot is specified via the *interpolation* field as one of either DISCRETE or

LINE_SEGMENT. Graph color is specified by the *color* field and is an integer index into a color table (each entry of the table consists of three short integers signifying the level of red, green, and blue components of the color). See section 3.2.2 for more about colors.

render The *render* field of the Graphic data structure is a pointer to routine which contains the drawing commands for its parent Graphic. In the case of a G_3D or INCREMENTAL Graphic this routine must be provided by the application for proper execution. In all other cases *render* is set internally.

R For the case of 2D plots, R describes the rectangle of bounds for the graph. It is of type *Rect* which has four floating point numbers as its elements (xm, xM, ym, yM), where (xm, ym) is the lower left hand corner of the rectangle and (xM, yM) is the upper right corner.

graph Again, *graph* is used only in the case that the Graphic contains a 2D plot. It is a data structure itself of type *Graph* and represents the 2D data of the Graphic in one of three ways: (i) in a link list of floating point pairs, (see the Appendix for an explanation of the link list utility), or (ii) as an array of floating point numbers versus its index, (iii) as an array of floating point numbers versus another array of floating point numbers. These three representations correspond directly to the Graphic types LINK_LIST, ARRAY, and ARRAY_vs_ARRAY given in Table 3. The structure definition is presented in Table 4. In the case that the Graphic type indicates an array type (i.e. ARRAY or ARRAY_vs_ARRAY), the data is represented in terms of the structure elements *n_pts, x* and *y*. In this case *n_pts* represents the number of points in the graph, *x* a pointer to the graph's abscissa data (if used) and *y*, a pointer to the graph's ordinate data.

```
typedef struct graph {
   Link_list *list;
   float *x, *y;
   int n_pts;
} Graph;
```

Table 4: Graph data structure

3.2.2 Color

Colors may be specified by three quantities indicating the level of red, green, and blue components. The following routines provide tools for the definition and

use of colors. Table 5 indicates a code fragment which illustrates the use of each routine.

color_entry(int index, int r, int g, int b)

Associates the positive integer *index* with the color specified by *r, g,* and *b.* This index is used by the routine **set_color()** and must be less than 256.

set_color(int index)

Sets the current drawing color to that of the entry at *index.* It is assumed that the entry has been set with the **color_entry()** routine.

map_interpolated_colors(float min, float max)

It is sometimes convenient to associate the magnitude of a quantity with a color. In the case that the quantity of interest represents temperature, associating dark blue with low temperatures and red with high temperatures is a natural scheme. Using this analogy a color mapping may be performed which associates the value *min* with BLUE and the value *max* with RED by invoking the routine **map_interpolated_colors**. Values between the two endpoints are interpolated smoothly.

interpolate_color_and_set(float val, float min, float max)

After color entries have been set by the routine **map_interpolated_colors** this routine may be used to set the drawing color to the color associated with *val.* It is assumed that *val* is between *min* and *max.*

3.2.3 Specification of Workspace

A typical application using the display manager would set up its workspace up with the structure of the skeleton shown in Table 6. The reader is refered to applications in Section 4 for explicit examples.

init_graph(Graphic *G, render)

A pointer to a Graphic, the *G* argument is assumed to have elements initialized to proper values before the call. The *render* argument is optional. It is the address of a function which will be called when its associated graph must be drawn.

```
#define REDISH 28    /* some integer between 0 and 255 */
{   float temperature;
                ⋮
   color_entry(REDISH, 240, 35, 55);
                ⋮
   set_color(REDISH)
      /* do some drawing with REDISH color */
                ⋮
   map_interpolated_colors(0.0, 100.0);
                ⋮       /* some temperature calculation */
   interpolate_color_and_set(temperature, 0.0, 100.0);
                ⋮
}
```

Table 5: Colors: An Example

```
Graphic G1,G2, ..., Gn;
                        ⋮
      set appropriate fields of G1
                        ⋮
init_graph(G1, my_render);
                        ⋮
       set appropriate fields of Gn
                        ⋮
init_graph(Gn);
```

Table 6: Window creation template

3.3 3D Model Manager

At the heart of GSMS, the model manager provides tools by which to describe complex dynamical 3D objects. Here a *dynamical* object is one whose 3D description (or state) changes over time. Unlike the case of modeling static 3D objects it is necessary to describe how a dynamical object changes. For instance a multi-link robot arm moving through space towards a target item is a dynamical object whose orientation and position of links determines its configuration at each time instant.

3.3.1 Describing a Dynamical Object

How to describe a dynamical object? This is the fundamental question which the model manager addresses. A complex object may be described in terms of its parts (or subobjects) and how they are connected. Consider the example of the

robot arm with two identical rigid links of given dimensions. Presumably the first link of the arm is fastened to a base or *ground.* This link is free to pivot around an axis perpendicular to the ground. Next the second link is attached to the first through a pivot connection allowing rotation only within a plane. Hence the arm has been completely described in terms of other composite static objects and their connections. These types of connections may be described by affine transformations (scalings, rotations, and translations) in three dimensional space. It is in this vain in which the manager models 3D objects. A detailed account follows.

3.3.2 The Object Network

To illustrate the idea of describing three dimensional objects in terms of affine combinations of subobjects, consider the case of developing an object model of a human being. One simple representation of a human is shown in Figure 3.

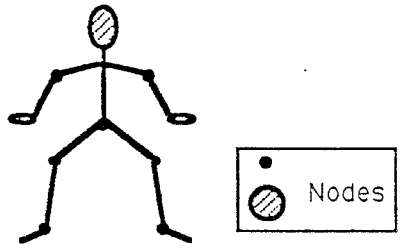


Figure 3: Stick model of a human being

The stick model in Figure 3 might be thought of as an *object network.* As in any network, the stick model network consists of nodes and branches. In the case of an object network nodes represent represent subobjects and/or primatives while branches represent connections between objects. All nodes in an object graph have a common ancestor or root which will be refered to as **'ground'**. It is the ground frame of reference in which GSMS renders all scenes. In Figure 3 one may think of the torso as being grounded.

3.3.3 Acyclical Object Networks

A limitation to the current implementation of the object manager is that it is only able to manipulate objects whose networks are acyclical. An acyclical object network is one whose graph has no closed loops. Objects with cyclical graphs require much higher level processing for the computation of thier graphical state than is currently supported. This area is a key focus of future expansion for GSMS.

Unless otherwise noted all network graphs encountered in the sequel will be assumed to be *acyclical.*

3.3.4 Connections

As mentioned earlier, connections between objects are described as affine transformations (scalings, rotations and translations) in three dimensional space. A detailed discussion of affine transformations and their implementation may be found in [4]. Consider the simple case of describing a ball and socket type connection between two objects, parent and child. The parent object is defined as the one which is closest to ground. These rotations and translations are performed relative to the parent object. A ball and socket connection implies full rotational freedom; therefore, this type of connection may be fully described as a rotation in three dimensional space. Realistically, though, the rotational freedom of the connection must be constrained in some way, e.g. to avoid collisions between parent and child objects. Constraining connections is left as the responsibility of the application.

Figure 4 is an example of a more informative representation of an object network. Here branches are replaced with labeled boxes associating a name with each branch (or subobject).

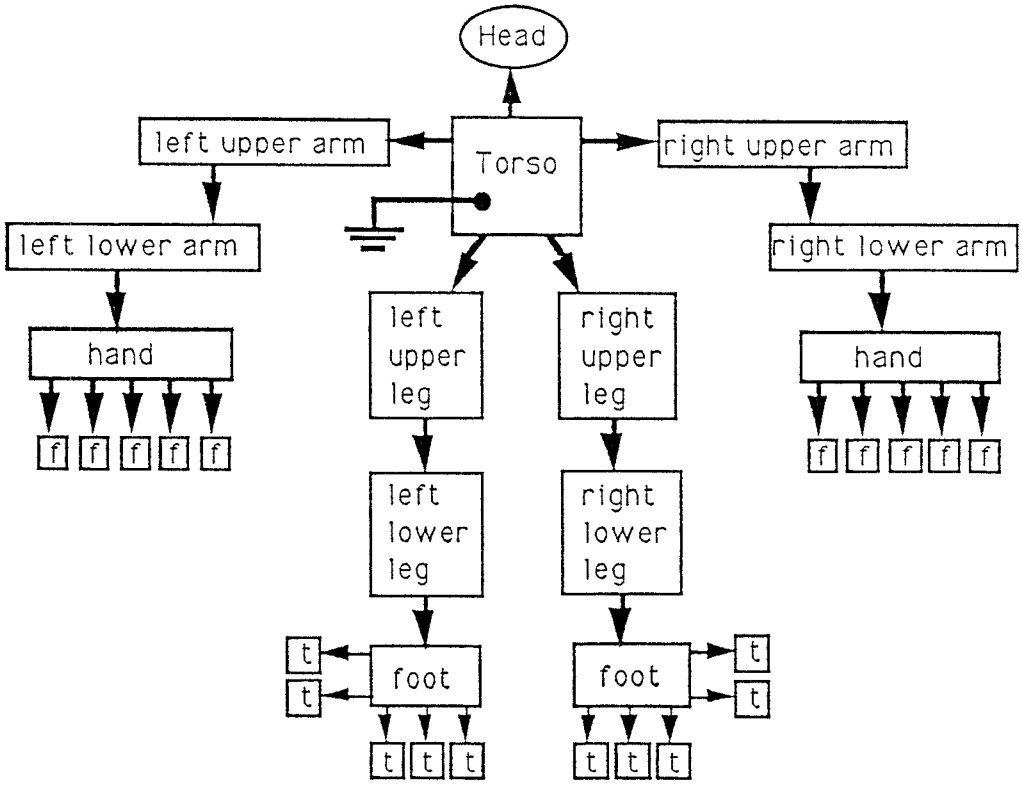


Figure 4: Object model of a human being

3.3.5 Object Primitives

The description of a 3D object proposed above implies a hierarchy; and therefore, the existence of a set of primative objects with which all other objects (or at

least objects of interest) may be constructed. One can always find such a primative set (take the set of all 3D objects); however, it is the set of primatives with the fewest elements which will be made the most useful. A more precise way to state this is the following: it is desired to find the smallest set of 3D graphical primitives whose span under affine spatial transformations covers the set of all complex objects of interest.

Among the 3D primative solids currently supported by GSMS are

1. cubes,
2. cylinders, and
3. spheres.

Since GSMS allows for affine transformation on any object in this primative set an entire class of objects may be generated, e.g. ellipsoids from spheres.

3.3.6 The Object_3D data structure

In order to implement the notion of an object network it is necessary to determine a representation for one of its elements, i.e. an object. This representation or data structure is given the name **Object_3D**. Displayed in Table 7 are the major fields of the data structure **Object_3D**. Because the fields of this structure are typically set via display manager routines for placing objects into a network, the application need not directly manipulate these fields. The reader is refered to the examples in Section 4.

```
typedef struct object_3d {
     struct object_3d *parent;
     struct object_3d *sub;
     struct object_3d *next;

     Transform *xform;

     int type;
     char *data; /* pointer to object_3d specific structure */

                 :
} Object_3d;
```

Table 7: The **Object_3D** data structure

parent, sub **and** *next* An acyclical network may be thought of as a tree like structure which starts at ground with a single root node. As a member of an acyclical object network, a node may have associated with it one or all of either (i) a parent node, (ii) a sub-node, or (iii) a next-node. For any arbitrary node in the network there is a unique parent node from which it is spawned; it may, however, have multiple siblings (nexts) as well as multiple children (subs). Note that because it belongs to an acyclical network, an object is connected to only its parent and its children (subs).

xform This field of the structure holds information regarding the connection (affine transformation) between this object and its parent object. Note that the field *xform* is itself a structure. The structure **Transform** is shown in table 8 and contains the nine quantities needed to uniquely determine an affine transformation. Scale, position, and rotation information are stored as vectors in $\mathbb{R}^3$ and may be accessed via the predefined constants **X,Y** and **Z**. For example to access the the y-coordinate of the position information of an associated object called *Ob* the correct reference would be *Ob->xform->position[Y]*.

```
typedef struct transform {
      float scale[3];
      float position[3];
      float rotation[3];
} Transform;
```

Table 8: The **Transform** data structure

type **and** *data* This field holds an integer value which indicates the type of object which the structure describes. A list of object types employed by GSMS is given in Table 9. The field *data* is a pointer to a structure containing the necessary data for the type.

object type	*Structure Name*	*description*
USER		application definable
CYLINDER	Cylinder	polygonal cylinder
SPHERE	Sphere	polygonal sphere
DEPTH_POLYGON	Depth_polygon	stretched polygon (along polygon normal)
POLYGON_LIST	Polygon_list	arbitrary polygonal object

Table 9: Object types

3.3.7 Instantiating an object

When describing a complex three dimensional object via the GSMS model manager, an application must first call the model manager routine **new_object()**. Besides providing the necessary initialization of the network graph it returns a pointer to the root object. Access to this pointer is essential in defining render routines (see the appendix for examples).

new_object()

Initializes and returns a pointer to the 'root' node of the object graph. Typically this routine is invoked only once at the begining of the object graph definition.

draw_object(Object_3d *object)

Draws the object tree of the complex object starting at the cell specified as its argument *object*. To render the entire complex object the application would pass the 'root' node address as the argument. The operation of this routine is detailed precisely in Section 3.3.9.

For each of the object types listed in Table 9 there is an associated routine which takes pertinent object geometry and materials information and places it into the appropriate data structure. In all cases these routines take as an argument a pointer to the appropriate data structure. It is the responsibility of the application to allocate the necessary memory to hold the structure. The operation and syntax of these routines is detailed in the following.

make_cylinder(Cylinder *c, float radius, height, int n_sides, top_material, bottom_material, edge_material_1, edge_material_2, char a)

Instantiates a polygonal approximation to a cylindrical shell with origin coincident with its centroid (assuming uniform mass) in the orientation aligned with the axis indicated in the argument $a \in \{x, y, z\}$. The resolution of the polygon approximation is controlled by the argument *n_sides* and signifies the number of rectangles which will comprise the walls of the cylinder. Arguments *edge_material_1* and *edge_material_2* indicate materials of alternating polygonal sections of the cylinder walls. All remaining arguments are self explanatory.

make_sphere(Sphere *s, float radius, int n_sides, material1, material2)

Instantiates a polygonal approximation to a spherical shell with origin at its center. Arguments *material1* and *material2* indicate materials of alternating polygonal sections of the sphere. The resolution of the polygon approximation is controlled by the argument *n_sides* and signifies the number of polygons lying on the equator of the sphere. Note if it is desired to have a sphere of uniform material one may simply indicate the same material for both *material1* and *material2*. All remaining arguments are self explanatory.

3.3.8 Some Modeling Routines

move_object_origin(Object_3D *Ob, float dx, dy, dz)

Incrementally adjusts the origin of the object *Ob* by the specified coordinates *dx, dy, dz.* Since it is the origin to which connections are assumed to be made, this routine provides control over the location of connections between objects. For instance, in the case of a two armed rigid robotic linkage modeled as two cylinders, object origins should occur at the base of the cylinders and not the default centroid location.

3.3.9 Rendering an Object (* application transparent)

Having introduced the concept of an object network, and consequently the **Object_3D** structure, the drawing of an object is easily accomplished. Specifically, one may apply a recursive algorithm to render any object described by an acyclical object network. Table 10 displays the C code instructions which GSMS uses for an object's rendering.

```
draw_object(O)
Object_3d *O;
{
      Object_3d *Ob;

      for( Ob=O; Ob != (Object_3d *) O; Ob = Ob->next) {
            pushmatrix();
                  render(Ob);
                  if (Ob->sub != (Object_3d *) O) draw_object(Ob->sub);
            popmatrix();
      }
}
```

Table 10: Drawing an object recursively

3.3.10 Forming the Object Network

GSMS provides two basic tools for constructing an object network **spawn_sub()** and **spawn_next()**. Note that object geometry information is referenced via a character pointer *data.* Hence whatever type data is stored at location *data* it should be type cast to the type **char** in a call to one of these routines. Described below is their syntax and operation.

Object_3D *spawn_sub(int type, char *data, Object_3D *parent, Transform *xform)

This routine enters an object into the network as a *subobject*, as the child of the object *parent*. It returns a pointer to the **Object_3D** element in which this object is placed. Initial connection information is given in the argument *xform*. It is a pointer to the transformation information which describes the connection between this new object and its parent. As such it is the responsibility of the application to allocate memory for the transformation information. This may be accomplished via the GSMS routine **new_transform()**; see below. Other information stored in the object network such as the object's *type* and a pointer to a block of *data* which describes the object are also required.

Object_3D *spawn_next(int type, char *data, Object_3D *from, Transform *xform)

This routine enters an object into the network as a *next-object*, an object on the same level of the object *from*. It returns a pointer to the **Object_3D** element in which this object is placed. Initial connection information is given in the argument *xform*. It is a pointer to the transformation information which describes the connection between this new object and its parent (the same as *from*'s parent). As such it is the responsibility of the application to allocate memory for the transformation information. This may be accomplished the GSMS routine **new_transform()**, see below. Other information stored in the object network such as the object's *type* and a pointer to a block of *data* which describes the object are also required.

Transform *new_transform(float sx, sy, sz, px, py, pz, rx, ry, rz)

Returning a pointer to a Transform structure, this routine allocates and stores the scale (sx,sy,sz), rotation (rx,ry,rz), and position (px,py,pz) information into a block of memory. Angles are assumed to be in degrees. It is most useful in conjunction with the **spawn_next()** and **spawn_sub()** routines.

3.4 Properties/Lighting Manager

The properties and lighting manager is the least developed of all the managers here and therefore the manager which could be most improved. It is provided in its current rudimentary form, however, to meet the main objective of GSMS: to provide a fast way to develop high quality graphical simulations. In this context 'high quality' refers to simulations which incorporate lighting and material models contributing to the resulting realism of the overall simulation.

Currently in GSMS there is only one lighting model which consists of two infinite white light sources: one hitting the x-y plane from above at an angle of 45° and on from below at the same angle. Another more grave drawback of the materials manager is that there are no management routines to deal with the creation and manipulation of user defined materials. [2] There is however a small set of predefined materials listed in Table 11 which may be accessed by the application.

[2] Providing more flexibility in the description of the lighting model as well as providing routines for the creation and management of user defined materials are key areas of future work on GSMS.

Name	*Description*
GREY_MAT	grey metallic material
GOLDEN_MAT	gold metallic material
RED_MAT	red metallic material
GREEN_MAT	green metallic material
BLUE_MAT	blue metallic material

Table 11: Predefined Materials

4 Applications

This section presents brief descriptions of three example GSMS applications. The appendix contains the example programs in full coded detail. Emphasis in these examples is placed on the graphical description and set-up of the application/user interface and ignores the details of the method of solving the dynamical system. All of the applications presented here are organized as two distinct files: (i) one which describes the physical three dimensional model (usually model.c) and (ii) one which describes the set-up of the workspace. Note the compact and succinct fashion in which potentially cumbersome and clumsy graphical simulations can be described with GSMS.

In all GSMS applications which include a 3-D window a navigation panel is automatically created. This navigation panel consists of three sliders which control the look at position (the user is always looking at the origin) via polar co-ordinates. Consequently the default labels for these sliders are 'distance', 'Pan X', and 'Pan Y'.

4.1 Simple Pendulum

Probably the best understood and most analyzed control system which exists is that of the simple pendulum. In this example a pendulum is modeled as an interconnection of three cylinders in opposing orientations. These cylinder objects are contained in a single branch of an acyclical object network.

4.2 Planet/Moon System

Planetary motion is another area in which relatively simple dynamical models may be developed. In this example a planet with two small moons in its orbit is modeled by three spheres in a three dimensional field of 'stars'. The three sphere objects are placed in a lateral object network, i.e. they are all siblings. The star field is implemented by providing an application defined render function which first calls the GSMS default render and then the star field render.

4.3 Elephant Trunk Manipulator

One of the most dextrous naturally occuring manipulators is that of the ele-

phant's trunk. Using GSMS it is especially simple to describe such a manipulator as one branch of an acyclical object network. Having described the elephant trunk so compactly, one may then focus his attention on its control. The application presented here involves a scheme for position control (code deleted). Modeling the elephant trunk as a series connection of n rigid links, one sees that there is a great many degrees of freedom with respect to position control. Having a three dimensional representation of a control algorithm's behavior can be a valuable source of information to the control engineer.

References

[1] B. W. Kernighhan and D. M. Ritchie, *The C Programming Language.* Prentice Hall, second ed., 1988.

[2] R. H. Byrne, "Interactive graphics and dynamical simulation in a distributed processing environment," Technical Report M.S. 90-7, University of Maryland Systems Research Center, 1990.

[3] D. A. Tristam and P. P. Walatka, "The panel library manual," Technical Report, NASA Ames Research Center and Sterling Software, 1989.

[4] J. Foley, A. vanDam, S. Feiner, and J. Hughes, *Computer Graphics Principles and Practice.* Addison Wesley, 1990.

Appendix (Coded Examples)

```
#
# Makefile for GSMS "pendulum" application      5/30/91 A.T.
#


PENDULUM.O =  pendulum.o model.o

IRIS = IRIS_4D

GSMS_DIR = /everest/users/tonyt

INCLUDE_DIR = $(GSMS_DIR)/include
LIB_DIR = $(GSMS_DIR)/lib

INCLUDE_DIRS= -I$(INCLUDE_DIR) -I/usr/local/include
LIB_DIRS= -L$(LIB_DIR) -L/usr/local/lib

GSMS_LIB = -lgraph -lpnl_mngr -llink_list -lpanel
LIBES = $(GSMS_LIB) -lgl -lm

CFLAGS = -D$(IRIS) $(INCLUDE_DIRS) $(LIB_DIRS)

pendulum: $(PENDULUM.O)
        $(CC) $(CFLAGS) $(PENDULUM.O) -o pendulum $(LIBES)
```

```
#include <gl.h>
#include <math.h>
#include "display.h"
#include "light.h"
#include "3d_tools.h"

#define ARM_RADIUS  0.1

/* pendulum angle */

float *theta_ptr;

/* base parameters */

float base_radius=0.3, base_width=0.1;
int base_sides = 10;

/* arm parameters */

float arm_radius=ARM_RADIUS, arm_width=2.0;
int arm_sides = 30;

/* weight parameters */

float weight_radius=0.4, weight_width=ARM_RADIUS;
int weight_sides = 20;

/* The objects */

Object_3d *the_ground, *base, *arm, *weight;

make_model()                                                    make_model
{
        int i;

        Cylinder base_cyl, arm_cyl,  weight_cyl;


/* Describe the parts: double pendulum made completely from cylinders */

/* base */

        make_cylinder(&base_cyl, base_radius, base_width, base_sides,
                                GREY_MAT, GREY_MAT, GREY_MAT, GREY_MAT, 'y');

/* arm */

        make_cylinder(&arm_cyl,
                                arm_radius,arm_width,arm_sides,
                                GREY_MAT, GREY_MAT, GREY_MAT, GREY_MAT, 'y');

/* weight */
```

```
        make_cylinder(&weight_cyl,
                                weight_radius,weight_width,weight_sides,
                                GREY_MAT, GREY_MAT, GREY_MAT, GREY_MAT, 'z');

/* Describe the connections, i.e. how the parts go together */
                                                                                60
        the_ground = new_object();

        base = spawn_next(CYLINDER, (char *) &base_cyl, the_ground,
                                        new_transform(
                                                1.0, 1.0, 1.0,
                                                0.0, arm_width, 0.0,
                                                0.0, 0.0, 0.0 )
                );

        arm = spawn_sub(CYLINDER, (char *) &arm_cyl, base,                      70
                                new_transform(
                                        1.0, 1.0, 1.0,
                                        0.0, -arm_width/2.0, 0.0,
                                        0.0, 0.0, 0.0 )
        );

        weight = spawn_sub(CYLINDER, (char *) &weight_cyl, arm,
                                new_transform(
                                        1.0, 1.0, 1.0,
                                        0.0, -(arm_width/2.0 + weight_radius), 0.0,     80
                                        0.0, 0.0, 0.0 )
        );

/* point to active model parameters */

        theta_ptr = &(base->xform->rotation[Z]);
}
```

```
#include <stdio.h>
#include <gl.h>
#include "math.h"
#include "display.h"
#include "light.h"
#include "pnl_manager.h"
#include "3d_tools.h"
#include "tools.h"
#include "graph.h"                                                    10

#define THETA_PTS 500

/* variables from model */

extern Object_3d *the_ground;
extern float *theta_ptr;

/* booleans */
                                                                      20
int automatic = 0, from_file = 0;

/* misc */

float frequency = 0.1, amplitude= 10, dummy, time = 0, time_increment=0.1;
float theta_data[THETA_PTS];
int theta_index=0;

/* file stuff */
                                                                      30
int n_data=0, index=0;
char in_filename[TYPE_IN_LEN];
Point2 data[20000];
FILE *fp;

char message[80]; /* buffer for report window message */

/* procedures */

void read_data(), reset_simulation();                                 40
int draw_scene();

/* Graphics */

Graphic Display, Theta;

main()                                                                main
{
        window_init();
        set_up_display();                                             50

        while (TRUE) {
                numerical_calculations();
```

```
                do_window_stuff();
        }
}

set_up_display()                                                    set_up_display
{
                                                                            60
/* windows */

        prefsize(600, 450);
        sprintf(Display.title, "Pendulum");
        Display.type = G_3D;
        init_graph(&Display, draw_scene);

        prefsize(600, 250);
        sprintf(Theta.title, "Theta");
        Theta.type = ARRAY;                                                 70
        Theta.graph.y = theta_data;
        Theta.graph.n_pts = theta_index;
        Theta.R = Box(0.0, (float) THETA_PTS, -180.0, 180.0);
        init_graph(&Theta);

/* 3D model */

        make_model();

/* panels (note: make_model() to be called first for proper initialization) */  80

        sprintf(in_filename, "No File"); /* initialize filename */

        new_panel("Simulation");
                make_variable("increment", &time_increment, null_function);
                make_type_in("file name", in_filename, read_data);
                make_toggle_button(" run", &from_file);
                make_button(" reset", reset_simulation);

        new_panel("Manual Control");                                        90
                make_toggle_button(" automatic", &automatic);

                make_slider("frequency", 0.0, 1.0, &frequency);
                make_slider("amplitude", 0.0, 270.0, &amplitude);

                make_slider("theta", -180.0, 180.0, theta_ptr); /* from model */
}

numerical_calculations() /* psuedo dynamics */                numerical_calculations
{                                                                           100
        int i;

        if (from_file) {
                for (i=index; data[i].x<time && index<n_data; i++) { }
                index=i;
                *theta_ptr = data[index].y;
```

```
                time += time_increment;
                update_theta_graph();

        } else if (automatic) {                                          110
                *theta_ptr = amplitude*sin( (double) (frequency*time) );
                time += time_increment;
                update_theta_graph();
        }
}

update_theta_graph()                                          update_theta_graph
{
        theta_data[theta_index] = *theta_ptr;
        theta_index = (theta_index+1)%THETA_PTS;                         120
        Theta.graph.n_pts = theta_index;
        Theta.new_data = TRUE;
}

void read_data()
{
        int i=0;

        if ( (fp = fopen(in_filename, "r")) == 0) {
                sprintf(message, "Unable to open file: '%s' ", in_filename);   130
                report_window(message);
                return;
        };

        while( fscanf(fp, "%f %f %f %f", &(data[i].x), &dummy, &dummy, &(data[i].y)) != EOF ) {
                i++;
        }
        n_data=i;

        printf("Read %d data points\n", n_data);                         140

        time = data[0].x;
        index=0;
}

void reset_simulation()
{
        time = data[0].x;
        *theta_ptr = data[0].y;
        index=0;                                                         150
}

draw_scene()                                                      draw_scene
{
        draw_axis();
        draw_object(the_ground);
}
```

```
#
# Makefile for GSMS "planet" application          5/30/91 A.T.
#


PLANET.O =    planet.o model.o

IRIS = IRIS_4D

GSMS_DIR = /everest/users/tonyt

INCLUDE_DIR = $(GSMS_DIR)/include
LIB_DIR = $(GSMS_DIR)/lib

INCLUDE_DIRS= -I$(INCLUDE_DIR) -I/usr/local/include
LIB_DIRS= -L$(LIB_DIR) -L/usr/local/lib

GSMS_LIB = -lgraph -lpnl_mngr -llink_list -lpanel
LIBES = $(GSMS_LIB) -lgl -lm

CFLAGS = -D$(IRIS) $(INCLUDE_DIRS) $(LIB_DIRS)

planet: $(PLANET.O)
        $(CC) $(CFLAGS) $(PLANET.O) -o planet $(LIBES)
```

```
#include <gl.h>
#include <math.h>
#include "display.h"
#include "light.h"
#include "3d_tools.h"

#define MAX_MOONS  10 /* maximum number of moons */

/* global parameters */                                                    10

float sphere_radius=1.0, sphere_sides=20, orbit_var=6.0, orbit[MAX_MOONS], n_moons=2;

/* data structures assigned to hold object information */

Sphere moon_sphere[MAX_MOONS], planet_sphere;

/* pointers to members of the object graph */

Object_3d *the_ground;    /* pointer to the ground (root) node */          20
Object_3d *planet, *moon[MAX_MOONS];


make_model()                                                    make_model
{
        int i;

/* initialize object graph */

        the_ground = new_object();                                         30

/* set up specific object information */

        update_model();

/* place main planet into object graph */

        planet = spawn_next(SPHERE, (char *) &planet_sphere, the_ground,
                                        new_transform(
                                                1.0, 1.0, 1.0,             40
                                                0.0, 0.0, 0.0,
                                                0.0, 15.0, 0.0 )
                );

/* place moons into object graph */

        for (i=0; i<n_moons; i++) spawn_moon(i);
}

update_model()                                                  update_model
{                                                                          51
        int i;
```

```
        for (i=0; i<n_moons; i++) {
                make_sphere(&moon_sphere[i],
                        0.4*(rand()%1000)/1000.*sphere_radius,
                        (int) sphere_sides, RED_MAT, RED_MAT);
        }
        make_sphere(&planet_sphere, sphere_radius,
                        (int) sphere_sides, BLUE_MAT, BLUE_MAT);                    60
}

spawn_moon(i)                                                          spawn_moon
int i;
{
        orbit[i] = orbit_var*(rand()%1000)/1000.+sphere_radius;
        moon[i] = spawn_next(SPHERE, (char *) &moon_sphere[i], the_ground,
                        new_transform(
                                1.0, 1.0, 1.0,
                                orbit[i], 0.0, 0.0,                             70
                                0.0, 0.0, 0.0 )
        );
}
```

```
#include <stdio.h>
#include <gl.h>
#include "math.h"
#include "display.h"
#include "light.h"
#include "pnl_manager.h"
#include "3d.h"
#include "3d_tools.h"
#include "tools.h"                                                    10
#include "graph.h"

#define TORAD(x) (x*M_PI/180.0)

/* externals from model */

extern Object_3d *the_ground, *planet, *moon[];
extern float sphere_radius, sphere_sides, n_moons, orbit_var, orbit[];

extern update_model();                                                20

/* dynamics parameters */

int i_time=0;
float time_step=1.0, rotation_scale = 10.0;
Point2 sincos[1000];

/* star background parameters */

float n_stars=1000.0, star_radius=0.05;                               30
Point3 star_list[100000];

void init_stars();
float get_rand();

/* Graphics */

Graphic Display, Theta;

int do_axis=FALSE;                                                    40
int draw_scene();

main()                                                                main
{
        init_stars();
        init_sincos();
        window_init();
        set_up_display();

        while (TRUE) {                                                50
                numerical_calculations();
                do_window_stuff();
        }
```

```
}

set_up_display()                                                    set_up_display
{

/* windows */
                                                                    60
        prefsize(700, 700);
        sprintf(Display.title, "Planet");
        Display.type = G_3D;
        init_graph(&Display, draw_scene);

/* 3D model */

        make_model();

/* panels (needs model definition) */                               70

        new_panel("Planet");
                make_toggle_button(" axis", &do_axis);
                make_variable("radius", &sphere_radius, update_model);
                make_variable("sides", &sphere_sides, update_model);
            new_column();
                make_variable("step", &time_step, null_function);
                make_variable("rot", &rotation_scale, null_function);

        new_panel("stars");                                         80
                make_variable("# stars", &n_stars, init_stars);
            new_column();
                make_variable("radius", &star_radius, null_function);

}


float t=0.0;

numerical_calculations() /* pseudo dynamics */                numerical_calculations
{                                                                   91
        int i;

        for (i=0; i<n_moons; i++) {
                moon[i]->xform->position[X] = orbit[i]* sincos[(i+1)*i_time%1000].y;
                moon[i]->xform->position[Z] = orbit[i]* sincos[(i+1)*i_time%1000].x;
                moon[i]->xform->rotation[Y] = rotation_scale*t;
        }

        planet->xform->rotation[Y] = rotation_scale*t/2.0;          100

        i_time += (int) time_step;
        i_time %= 1000;
        t += time_step;
}
```

init_sincos

```
init_sincos()
{
        int i=0;
        float x;

        for (x=0.0; x<2*M_PI; x+= 2*M_PI/1000.0) {
                sincos[i].x = sin(x);
                sincos[i++].y = cos(x);
        }
}
```

draw_scene

```
draw_scene()
{
        if (do_axis) draw_axis();
        draw_stars();
        draw_object(the_ground);
}

void init_stars()
{
        int i;
        float radius, theta, phi;

        radius= FAR/3.0;

        for (i=0; i<n_stars; i++) {
                theta = M_PI*get_rand();
                phi = M_PI*get_rand();

                star_list[i].x = radius*sin(theta)*cos(phi);
                star_list[i].y = radius*sin(phi);
                star_list[i].z = radius*cos(theta)*cos(phi);
        }
}

float get_rand()
{
        return( 1.0 - 2.0*(rand() % 1000)/1000.0 );
}
```

draw_stars

```
draw_stars()
{
        int i;

        RGBcolor(200, 220, 250);
        for (i=0; i<n_stars; i++) {
                star(star_list[i], star_radius);
        }
}
```

```
#
# Makefile for GSMS "elephant" application      5/30/91 A.T.
#

ELEPHANT.O =  elephant.o model.o

IRIS = IRIS_4D

GSMS_DIR = /everest/users/tonyt                                    10

INCLUDE_DIR = $(GSMS_DIR)/include
LIB_DIR = $(GSMS_DIR)/lib

INCLUDE_DIRS= -I$(INCLUDE_DIR) -I/usr/local/include
LIB_DIRS= -L$(LIB_DIR) -L/usr/local/lib

GSMS_LIB = -lgraph -lpnl_mngr -llink_list -lpanel
LIBES = $(GSMS_LIB) -lgl -lm
                                                                   20
CFLAGS = -D$(IRIS) $(INCLUDE_DIRS) $(LIB_DIRS)

elephant: $(ELEPHANT.O)
        $(CC) $(CFLAGS) $(ELEPHANT.O) -o elephant $(LIBES)
```

```
#include <gl.h>
#include <math.h>
#include "display.h"
#include "light.h"
#include "3d_tools.h"
#include "elephant.h"

/* Note: This code is generalized to generate N legged creatures with each leg
        described by N_JOINTS number of rigid links. To retrieve the elephant
        trunk simply define N_LEGS as 1.
*/

float *jx[N_LEGS][N_JOINTS], *jy[N_LEGS][N_JOINTS], *jz[N_LEGS][N_JOINTS];
float length[N_JOINTS];

float radius=0.3, height=1.2;
float n_sides = 30;

Object_3d *the_ground, *Head, *current_O;
Cylinder cyl;
Sphere head;

make_model()                                                    make_model
{
        int i,j;
        double x;

        make_sphere(&head, 3.3*radius, 40, GREY_MAT, GREY_MAT);
        make_left_cylinder(&cyl, 0.5, height, (int) n_sides,
                                GREY_MAT, GREY_MAT, GREY_MAT, GREY_MAT, 'z');

        the_ground = new_object();

        Head = spawn_next(SPHERE, (char *) &head, the_ground,
                                        new_transform(
                                                1.0, 1.8, 1.2,
                                                0.0, 0.0, 0.0,
                                                0.0, 0.0, 0.0 )
                );

        for (j=0; j<N_LEGS; j++) {
                i=0;
                x = 2.0*3.1415*j/((float) N_LEGS);
                spawn_sub(CYLINDER, (char *) &cyl, Head,
                                        new_transform(
                                                0.95, 0.95, 0.85,
                                                sin(x), 0.0, cos(x),
                                                10.0, 360.0*j/((float) N_LEGS), 0.0)
                );
                jx[j][i] = &current_O->xform->rotation[X];
                jy[j][i] = &current_O->xform->rotation[Y];
                jz[j][i] = &current_O->xform->rotation[Z];
```

```
        for (i=1; i<N_JOINTS; i++) {

                spawn_sub(CYLINDER, (char *) &cyl, current_O,
                                new_transform(
                                        0.95, 0.95, 0.85,
                                        0.0, 0.0, 1.1,
                                        10.0, 0.0, 0.0 )
                );
                jx[j][i] = &current_O->xform->rotation[X];
                jy[j][i] = &current_O->xform->rotation[Y];
                jz[j][i] = &current_O->xform->rotation[Z];
        }
    }

}
```

```
#include <stdio.h>
#include <math.h>
#include <gl.h>
#include "math.h"
#include "display.h"
#include "light.h"
#include "3d_tools.h"
#include "tools.h"
#include "graph.h"                                                    10
#include "elephant.h"

/* variables from model */

extern Object_3d *the_ground, *Head;
extern float radius, height, n_sides;
extern float *jx[N_LEGS][N_JOINTS], *jy[N_LEGS][N_JOINTS], *jz[N_LEGS][N_JOINTS];
float length[20];

/* booleans */                                                        20

int x_rot=FALSE, y_rot=FALSE;

/* misc */

int mode=SIN;
float step=1.0, freq = 0.1, star_diameter=0.2;
Point3 target;

/* procedures */                                                      30

int draw_scene(), reset_simulation();

/* Graphics */

Graphic Display;

main()                                                                main
{
        window_init();                                                40
        set_up_display();

        while(TRUE) {
                numerical_calculations();
                do_window_stuff();
        }
}

set_up_display()                                                      set_up_display
{                                                                     50
        int i,j;
        char *s_ptr;
```

```
        prefsize(800, 650);
        sprintf(Display.title, "Elephant");
        Display.type = G_3D;
        init_graph(&Display, draw_scene);

        target.x = 5.0;
        target.y = 0.0;                                                    60
        target.z = 0.0;

        make_model();

        new_panel("Control");
                make_slider("step", 1.0, 10.0, &step);
                make_slider("freq", 0.01, 0.09, &freq);
           new_column();
                make_toggle_button(" y_rot", &y_rot);
                make_toggle_button(" x_rot", &x_rot);                      70
                make_button(" reset", reset_simulation);

        new_panel("Transform");
                make_slider("head y", −360.0, 360.0, &Head−>xform−>rotation[Y]);
                for (i=0; i<N_JOINTS; i++) {
                        s_ptr = (char *) malloc(30);
                        sprintf(s_ptr, "joint #%d", i+1);
                        make_slider(s_ptr, −30.0, 50.0, jx[0][i]);
                }
                                                                           80
        new_panel("Target");
                make_variable("length", &star_diameter, null_function);
                make_slider("x", −10.0, 10.0, &target.x);
                make_slider("y", −10.0, 10.0, &target.y);
                make_slider("z", −10.0, 10.0, &target.z);
}

double t;

numerical_calculations() /* position control?  HA! (for exhibition only) */numerical_calculations
{                                                                          91
        int i,j;

        for (j=0; j<N_LEGS; j++)
        for (i=0; i<N_JOINTS; i++) {

                switch(mode) {

                case RANDOM:
                  if (x_rot) *jx[j][i] −= step*( (rand()% 1000)/1000.0 − 0.5);   100
                  if (y_rot) *jy[j][i] −= step*( (rand()% 1000)/1000.0 − 0.5);
                  break;

                case SIN:
                  if (x_rot) *jx[j][i] −= step*sin(freq*t);
                  if (y_rot) *jy[j][i] −= step*cos(freq*t);
```

```
                t += 0.1;
                break;
            }
        }
}
```

reset_simulation

```
reset_simulation()
{
        int i,j;

        for (j=0; j<N_LEGS; j++) {
                i=0;
                *jy[j][i] = j*360.0/((float) N_LEGS);
                for (i=1; i<N_JOINTS; i++) {
                        *jx[j][i] = 10.0;
                        *jy[j][i] = 0.0;
                }
        }
}
```

draw_scene

```
draw_scene()
{
        draw_axis();
        draw_object(the_ground);
        draw_target();
}
```

draw_target

```
draw_target()
{
        RGBcolor(100, 100, 220);
        star(target, star_diameter);
}
```